# PGI Accelerator™ Compilers
# OpenACC Getting Started Guide

Version 12.5

The Portland Group®

OpenACC 2012 Getting Started Guide


Printed in the United States of America

First Printing: Release 2012, version 12.3, March 2012
Second Printing: Release 2012, version 12.4, April 2012
Third Printing: Release 2012, version 12.5, May 2012

Technical support: trs@pgroup.com
Sales: sales@pgroup.com
Web: www.pgroup.com

# Contents

CHAPTER 1

# Overview

The OpenACC Application Program Interface is a collection of compiler directives and runtime routines that allow you, the programmer, to specify loops and regions of code in standard C and Fortran that you want offloaded from a host CPU to an attached accelerator, such as a GPU. The OpenACC API was designed and is maintained by an industry consortium. See the OpenACC website http://www.openacc.org for more information about the OpenACC API. In particular, the whole specification is available at http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

This Getting Started guide will help you prepare your system for using the PGI OpenACC implementation, and will give examples of how to write, build and run programs using the OpenACC directives. More information about PGI's OpenACC implementation is available at http://www.pgroup.com/openacc.

## Terminology and Definitions

Throughout this document certain terms have very specific meaning:

- OpenACC is the name of the specification, which includes compiler directives, runtime routines, and environment variables.
- PGCC and PGFORTRAN are the names of the PGI compiler products.
- `pgcc` and `pgfortran` are the names of the PGI compiler drivers. `pgfortran` may also be spelled `pgf90` and `pgf95`.
- CUDA is the parallel computing platform and programming model invented and supported by NVIDIA for GPUs.

# System Prerequisites

Using this release of PGI OpenACC API implementation requires the following:

- A 32-bit or 64-bit Linux or Microsoft Windows Intel or AMD x86 system, with a PGI-supported and CUDA-supported release of the operating system. Get information about the PGI-supported Linux releases at http://www.pgroup.com/support/install.htm. Get information about CUDA-supported Linux releases at http://www.nvidia.com/cuda.
- A CUDA-enabled NVIDIA GPU.
- An installed CUDA driver, version 4.0 or later. CUDA drivers can be downloaded at http://www.nvidia.com/cuda.

# Prepare Your System

To enable OpenACC, follow these steps:

1. Download the latest 12.5 Linux packages from the PGI website at http://www.pgroup.com/support/downloads.php
2. Install the downloaded package.
3. Put the installed bin directory on your path.
4. Run `pgaccelinfo` to see that your NVIDIA GPU and CUDA drivers are properly installed and available. You should see output that looks something like the following:

```
CUDA Driver Version:           4010
NVRM version: NVIDIA UNIX x86_64 Kernel Module  285.05.33  Thu Jan
19 14:07:02 PST 2012

Device Number:                 0
Device Name:                   GeForce GTX 280
Device Revision Number:        1.3
Global Memory Size:            1073414144
Number of Multiprocessors:     30
Number of Cores:               240
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 16384
Registers per Block:           16384
Warp Size:                     32
Maximum Threads per Block:     512
Maximum Block Dimensions:      512, 512, 64
Maximum Grid Dimensions:       65535 x 65535 x 1
Maximum Memory Pitch:          2147483647B
Texture Alignment:             256B
Clock Rate:                    1296 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            No
ECC Enabled:                   No
Memory Clock Rate:             1107 MHz
Memory Bus Width:              512 bits
Max Threads Per SMP:           1024
Async Engines:                 1
Unified Addressing:            No
Initialization time:           856523 microseconds
Current free memory:           1015942912
Upload time (4MB):             8523 microseconds (4459 ms pinned)
Download time:                 15105 microseconds (4558 ms pinned)
Upload bandwidth:              492 MB/sec ( 940 MB/sec pinned)
Download bandwidth:            277 MB/sec ( 920 MB/sec pinned)
```

This tells you the CUDA driver version, the name and compute capability of the GPU (or GPUs, if you have more than one), the available memory, and so on.

## Supporting Documentation and Examples

You may want to consult the OpenACC 1.0 specification, included with this release, for additional information. It is also available at the OpenACC website, http://www.openacc-standard.org. Simple examples appear in Chapter 3, Using OpenACC with the PGI Compilers.

An SDK will be available at the OpenACC website. In future releases, it will be installed with the PGI compilers, at /opt/pgi/linux86[-64]/2012/openacc/SDK.

CHAPTER 2

# About the PGI OpenACC Beta

PGI Release 2012 version 12.5 includes partial support for the OpenACC directive-based GPU accelerator programming standard version 1.0. A list of the features supported in this version can be found Chapter 5.

Use of the OpenACC capabilities in this version are restricted under the terms of the PGI Beta Test License Agreement (the "BTLA"), a copy of which is included with your installation. Please read the BTLA before using the OpenACC features. Note specifically you must agree that:

1. The Software will be used for the express and sole purpose of evaluation toward determination of suitability for purchase of an upcoming production release of the Software.
2. In the absence of explicit permission from STMicroelectronics, The Portland Group, performance results obtained using the Software will not be published or presented in a public forum.

CHAPTER 3

# Using OpenACC with the PGI Compilers

The OpenACC directives are enabled by adding the `-acc` flag to the PGI compiler command line. This release targets OpenACC to NVIDIA GPUs. See Chapter 3 for discussion about using OpenACC directives or the `-acc` flag with PGI Accelerator directives, or with object files compiled with the PGI Accelerator directives. In particular, specifying `-acc` with or without `-ta=nvidia` will enable the OpenACC directives and the OpenACC runtime.

This release does not fully implement the OpenACC 1.0 specification. See Chapter 4 for details about what features are included in this release, and what features will be coming in updates over the next few months.

## C Examples

The simplest C example of OpenACC would be a vector addition on the GPU:

```
#include <stdio.h>
#include <stdlib.h>

void vecaddgpu( float *restrict r, float *a, float *b, int n ){
    #pragma acc kernels for copyin(a[0:n],b[0:n]) copyout(r[0:n])
    for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
    int n;   /* vector length */
    float * a;  /* input vector 1 */
    float * b;  /* input vector 2 */
    float * r;  /* output vector */
    float * e;  /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else  n = 100000;  /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
        a[i] = (float)(i+1);
        b[i] = (float)(1000*i);
    }
    /* compute on the GPU */
    vecaddgpu( r, a, b, n );
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
        if( r[i] != e[i] ){
            ++errs;
        }
    }
    printf( "%d errors found\n", errs );
    return errs;
}
```

The important part of this example is the routine `vecaddgpu`, which includes one OpenACC directive for the loop. This (`#pragma acc`) directive tells the compiler to generate a kernel for the following loop (`kernels for`), to allocate and copy from the host memory into the GPU memory n elements for the vectors a and b before executing on the GPU, starting at a[0] and b[0] (`copyin(a[0:n],b[0:n])`), and to allocate n

elements for the vector r before executing on the GPU, and copy from the GPU memory out to the host memory those n elements, starting at r[0] (`copyout(r[0:n])`).

If you type this example into a file `a1.c`, you can build it with this release using the command `pgcc -acc a1.c`. The `-acc` flag enables recognition of the OpenACC pragmas and includes the OpenACC runtime library. This will generate the usual a.out executable file, and you run the program by running a.out as normal. You should see the output:

```
0 errors found
```

If instead you get the output

```
libcuda.so not found, exiting
Please check that the CUDA driver is installed and the shared object
is in the install directory or on your LD_LIBRARY_PATH.
```

then there is something wrong with your hardware installation or your CUDA driver.

You can enable additional output by setting environment variables. If you set the environment variable `ACC_NOTIFY` to a positive integer value, then the runtime will print a line of output each time you run a kernel on the GPU. For this program, you might get output that looks like:

```
 launch kernel  file=/user/guest/guide/a1.c function=vecaddgpu
line=6 device=0 grid=391 block=256
0 errors found
```

The extra output tells you that the program launched a kernel for the loop at line 6, with a CUDA grid of size 391, and a thread block of size 256.

If you set the environment variable `PGI_ACC_TIME` to a positive integer value, the runtime will summarize the time taken for data movement between the host and GPU, and computation on the GPU. For this program, you might get output like:

```
0 errors found

Accelerator Kernel Timing data
/home/mwolfe/OpenACC/guide/a1.c
  vecaddgpu
    5: region entered 1 time
        time(us): total=1051789 init=1047282 region=4507
                  kernels=53 data=3710
        w/o init: total=4507 max=4507 min=4507 avg=4507
        6: kernel launched 1 times
            grid: [391]  block: [256]
            time(us): total=53 max=53 min=53 avg=53
```

This tells you that the program entered one accelerator region and spent a total of about 1 second in that region. Most of that time was spent initializing the GPU (refer to

Troubleshooting Tips and Known Limitations). The program spent 3.7 milliseconds moving data between the host and GPU, and 53 microseconds executing the kernel. It also summarizes the time spent in each kernel generated from the OpenACC compute regions.

You will also find it useful to enable the compiler feedback when you are writing your own OpenACC programs. This is enabled with the `-Minfo` flag. If we compile this program with the command `pgcc -acc -fast -Minfo a1.c`, we will get the output:

```
vecaddgpu:
      5, Generating copyout(r[:n])
         Generating copyin(a[:n])
         Generating copyin(b[:n])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      6, Loop is parallelizable
         Accelerator kernel generated
          6, #pragma acc loop gang, vector(256) /* blockIdx.x
threadIdx.x */
             CC 1.0 : 5 registers; 60 shared, 4 constant, 0 local
memory bytes; 100% occupancy
             CC 2.0 : 8 registers; 8 shared, 68 constant, 0 local
memory bytes; 100% occupancy
```

This tells you that the compiler generated two versions of the code, one for NVIDIA devices with compute capability 1.0 and higher, and one for devices with compute capability 2.0 and higher. It also gives the *schedule* used for the loop; in this case ,the schedule is `gang, vector(256)`. This means the iterations of the loop are broken into vectors of 256, and the vectors executed in parallel by SMPs of the GPU.

This output is important because it tells you when you are going to get parallel execution or sequential execution. If you remove the restrict keyword from the declaration of the dummy argument r to the routine `vecaddgpu`, the `-Minfo` output will tell you that there may be dependences between the stores through the pointer r and the fetches through the pointers a and b:

```
      6, Complex loop carried dependence of '*(b)' prevents
parallelization
         Complex loop carried dependence of '*(a)' prevents
parallelization
         Loop carried dependence of '*(r)' prevents parallelization
         Loop carried backward dependence of '*(r)' prevents
vectorization
         Accelerator kernel generated
          6, #pragma acc loop seq
```

The compiler generated a sequential kernel, which will run about 1000 times slower than the parallel kernel. For this simple program, the total time is dominated by GPU initialization, so you might not notice, but in production mode you need parallel kernel execution to get acceptable performance.

For our second example, we will modify the program slightly by replacing the data clauses on the kernels pragma with a `present` clause, and add a data construct surrounding the call to the `vecaddgpu` routine. The data construct moves the data across to the GPU in the main program. The present clause in the `vecaddgpu` routine tells the compiler to use the GPU copy of the data that has already been allocated on the GPU. If you run this program on the GPU with PGI_ACC_TIME set, you will see that the kernel region now has no data movement associated with it. Instead, the data movement is all associated with the data construct in the main program.

```
#include <stdio.h>
#include <stdlib.h>

void vecaddgpu( float *restrict r, float *a, float *b, int n ){
    #pragma acc kernels for present(r,a,b)
    for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
    int n;   /* vector length */
    float * a;  /* input vector 1 */
    float * b;  /* input vector 2 */
    float * r;  /* output vector */
    float * e;  /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else  n = 100000;  /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
        a[i] = (float)(i+1);
        b[i] = (float)(1000*i);
    }
```

```
    /* compute on the GPU */
    #pragma acc data copyin(a[0:n],b[0:n]) copyout(r[0:n])
    {
        vecaddgpu( r, a, b, n );
    }
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
        if( r[i] != e[i] ){
            ++errs;
        }
    }
    printf( "%d errors found\n", errs );
    return errs;
}
```

# Fortran Examples

The simplest Fortran example of OpenACC would be a vector addition on the GPU:

```
module vecaddmod
 implicit none
contains
 subroutine vecaddgpu( r, a, b, n )
  real, dimension(:) :: r, a, b
  integer :: n
  integer :: i
!$acc kernels do copyin(a(1:n),b(1:n)) copyout(r(1:n))
  do i = 1, n
   r(i) = a(i) + b(i)
  enddo
 end subroutine
end module

program main
 use vecaddmod
 implicit none
 integer :: n, i, errs, argcount
 real, dimension(:), allocatable :: a, b, r, e
 character*10 :: arg1
 argcount = command_argument_count()
 n = 1000000  ! default value
 if( argcount >= 1 )then
  call get_command_argument( 1, arg1 )
  read( arg1, '(i)' ) n
  if( n <= 0 ) n = 100000
 endif
```

```
allocate( a(n), b(n), r(n), e(n) )
 do i = 1, n
  a(i) = i
  b(i) = 1000*i
 enddo
 ! compute on the GPU
 call vecaddgpu( r, a, b, n )
 ! compute on the host to compare
 do i = 1, n
  e(i) = a(i) + b(i)
 enddo
 ! compare results
 errs = 0
 do i = 1, n
  if( r(i) /= e(i) )then
    errs = errs + 1
  endif
 enddo
 print *, errs, ' errors found'
 if( errs ) call exit(errs)
end program
```

The important part of this example is the subroutine `vecaddgpu`, which includes one OpenACC directive for the loop. This (!$acc) directive tells the compiler to generate a kernel for the following loop (`kernels do`), to allocate and copy from the host memory into the GPU memory n elements for the vectors a and b before executing on the GPU, starting at a(1) and b(1) (`copyin(a(1:n),b(1:n)`), and to allocate n elements for the vector r before executing on the GPU, and copy from the GPU memory out to the host memory those n elements, starting at r(1) (`copyout(r(1:n)`).

If you type this example into a file `f1.f90`, you can build it with this release using the command `pgfortran -acc f1.f90`. The `-acc` flag enables recognition of the OpenACC pragmas and includes the OpenACC runtime library. This will generate the usual a.out executable file, and you run the program by running a.out as normal. You should see the output:

```
0 errors found
```

If instead you get the output

```
libcuda.so not found, exiting
Please check that the CUDA driver is installed and the shared object
is in the install directory or on your LD_LIBRARY_PATH.
```

then there is something wrong with your hardware installation or your CUDA driver.

You can enable additional output by setting environment variables. If you set the environment variable `ACC_NOTIFY` to a positive integer value, then the runtime will print a line of output each time you run a kernel on the GPU. For this program, you might get output that looks like:

```
 launch kernel  file=/user/guest/guide/f1.f90 function=vecaddgpu
line=9 device=0 grid=391 block=256
0 errors found
```

The extra output tells you that the program launched a kernel for the loop at line 9, with a CUDA grid of size 391, and a thread block of size 256.

If you set the environment variable `PGI_ACC_TIME` to a positive integer value, the runtime will summarize the time taken for data movement between the host and GPU, and computation on the GPU. For this program, you might get output like:

```
0 errors found

Accelerator Kernel Timing data
/home/mwolfe/OpenACC/guide/f1.f90
  vecaddgpu
    8: region entered 1 time
        time(us): total=1016881 init=1013441 region=3440
                  kernels=53 data=2484
        w/o init: total=3440 max=3440 min=3440 avg=3440
        9: kernel launched 1 times
            grid: [391]  block: [256]
            time(us): total=53 max=53 min=53 avg=53
```

This tells you that the program entered one accelerator region and spent a total of about 1 second in that region. Most of that time was spent initializing the GPU (refer to the section Troubleshooting Tips and Known Limitations). The program spent 2.4 milliseconds moving data between the host and GPU, and 53 microseconds executing the kernel. It also summarizes the time spent in each kernel generated from the OpenACC compute regions.

You will also find it useful to enable the compiler feedback when you are writing your own OpenACC programs. This is enabled with the `-Minfo` flag. If we compile this program with the command `pgcc -acc -fast -Minfo a1.c`, we will get the output:

```
vecaddgpu:
      8, Generating copyin(b(:n))
         Generating copyin(a(:n))
         Generating copyout(r(:n))
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      9, Loop is parallelizable
         Accelerator kernel generated
          9, !$acc loop gang, vector(256) ! blockidx%x threadidx%x
             CC 1.0 : 4 registers; 72 shared, 4 constant, 0 local
memory bytes; 100% occupancy
             CC 2.0 : 10 registers; 8 shared, 80 constant, 0 local
memory bytes; 100% occupancy
```

This tells you that the compiler generated two versions of the code, one for NVIDIA devices with compute capability 1.0 and higher, and one for devices with compute capability 2.0 and higher. It also gives the *schedule* used for the loop; in this case ,the schedule is `gang, vector(256)`. This means the iterations of the loop are broken into vectors of 256, and the vectors executed in parallel by SMPs of the GPU. This output is important because it tells you when you are going to get parallel execution or sequential execution.

For our second example, we will modify the program slightly by replacing the data clauses on the kernels pragma with a `present` clause, and add a data construct surrounding the call to the `vecaddgpu` subroutine. The data construct moves the data across to the GPU in the main program. The present clause in the `vecaddgpu` subroutine tells the compiler to use the GPU copy of the data that has already been allocated on the GPU. If you run this program on the GPU with `PGI_ACC_TIME` set, you will see that the kernel region now has no data movement associated with it. Instead, the data movement is all associated with the data construct in the main program.

In Fortran programs, you don't have to specify the array bounds in data clauses, if the compiler can figure out the bounds from the declaration, or if the arrays are assumed-shape dummy arguments or allocatable arrays.

```
module vecaddmod
 implicit none
contains
 subroutine vecaddgpu( r, a, b, n )
  real, dimension(:) :: r, a, b
  integer :: n
  integer :: i
!$acc kernels do present(r,a,b)
  do i = 1, n
   r(i) = a(i) + b(i)
  enddo
 end subroutine
end module

program main
 use vecaddmod
 implicit none
 integer :: n, i, errs, argcount
 real, dimension(:), allocatable :: a, b, r, e
 character*10 :: arg1
 argcount = command_argument_count()
 n = 1000000  ! default value
 if( argcount >= 1 )then
  call get_command_argument( 1, arg1 )
  read( arg1, '(i)' ) n
  if( n <= 0 ) n = 100000
 endif
 allocate( a(n), b(n), r(n), e(n) )
 do i = 1, n
  a(i) = i
  b(i) = 1000*i
 enddo
 ! compute on the GPU
!$acc data copyin(a,b) copyout(r)
   call vecaddgpu( r, a, b, n )
!$acc end data
 ! compute on the host to compare
 do i = 1, n
  e(i) = a(i) + b(i)
 enddo
 ! compare results
 errs = 0
 do i = 1, n
  if( r(i) /= e(i) )then
    errs = errs + 1
  endif
 enddo
 print *, errs, ' errors found'
 if( errs ) call exit(errs)
end program
```

# Troubleshooting Tips and Known Limitations

This release of the PGI compilers does not implement the full OpenACC specification. For an explanation of what features are not yet implemented, refer to Chapter 5, Implemented Features.

The Linux CUDA driver will power down an idle GPU. This means if you are using a GPU with no attached display, or an NVIDIA Tesla compute-only GPU, and there are no open CUDA contexts, the GPU will power down until it is needed. Since it takes about a second to power the GPU back up, you may experience noticeable delays when you start your program. When you run your program with the environment variable PGI_ACC_TIME set to 1, this time will appear as initialization time. If you have an NVIDIA S1070 or S2050 with four GPUs, this initialization time may be up to 4 seconds. If you are running many tests, or want to isolate the actual time from the initialization time, you can run the PGI utility `pgcudainit` in the background. This will open a CUDA context and hold it open until you kill it or let it compete.

This release does not allow a `present` data clause that specifies a C pointer which also appears in a lexically enclosing `data` construct, if the pointer value has changed.

In this release, the compiler will not automatically use the NVIDIA shared memory as a data cache when the `-acc` flag is set.

CHAPTER 4

# PGI Accelerator Model Interoperability

This chapter describes how the PGI OpenACC implementation and object files created with the PGI OpenACC compilers interoperate with the PGI Accelerator model, and object files created with the PGI Accelerator compilers. PGI continues to support the PGI Accelerator model and directives, and will evolve this model to be fully compatible with and to extend OpenACC.

## OpenAcc New Features

OpenACC has added five important features that were not available in the PGI Accelerator model.

- OpenACC has two types of compute constructs, the `acc parallel` construct and the `acc kernels` construct. The acc kernels construct is very similar to the PGI Accelerator `acc region` construct. The `acc parallel` construct is new to OpenACC. This release of the PGI OpenACC compilers has limited support for the `acc parallel` construct. It only supports this construct with a single contained loop with the `gang` clause, and only when contained work-sharing loops are also tightly nested.
- OpenACC has the `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout`, and `present_or_create` data clauses. These give functionality that is similar to the `reflected` data clause in the PGI Accelerator model, except the `present` clauses can be used for global data, and do not need an explicit interface. This release of the PGI OpenACC compilers implements the `present` clauses when the `-acc` flag is enabled.
- OpenACC has support for asynchronous data movement between the host and GPU, and asynchronous computation on the GPU, using the `async` clause and the `wait` directive. This release of the PGI OpenACC compilers implements the

`async` clause and asynchronous data movement and computation, and the `wait` directive. The `async` clause is accepted on all relevant OpenACC directives, as well as PGI Accelerator directives.

- OpenACC defines three levels of parallelism: gang, worker and vector. The PGI Accelerator model defines two levels of parallelism: parallel and vector, where each level can have multiple dimensions. The OpenACC gang parallelism corresponds directly to the PGI Accelerator parallel level of parallelism. The OpenACC vector parallelism corresponds to the PGI Accelerator vector parallelism. The OpenACC worker parallelism is new. This release of the PGI OpenACC compilers does not implement worker parallelism.
- OpenACC supports an explicit `reduction` clause for inner loops. This release of the PGI OpenACC compilers does not support the `reduction` clause.

There are other differences between OpenACC and the PGI Accelerator model, described in the following sections.

# PGI Accelerator Features not available in the OpenACC

The PGI Accelerator Model has four features that are not available in OpenACC.

- In Fortran, the PGI Accelerator model has `mirror` and `reflected` data clauses. The `mirror` data clause tells the compiler to allocate a device copy of the array whenever the host copy is allocated. For global arrays, the device copy is accessible in any subprogram where the host copy is accessible. The `reflected` data clause tells the compiler to pass the address of the device copy of an array as an argument when it passes the address of the host copy of the array. The functionality of both of these is similar to that of the `present` data clause in OpenACC. The advantage of `mirror` and `reflected` is that the compiler can check at compile time whether the calling routine is passing an array with a device copy, or whether the global array has a device copy. The advantage of the `present` clause is that it doesn't require an explicit interface, and the same mechanism can be applied to dummy arguments and global arrays.
- The PGI Accelerator model version 1.x allows for any rectangular subarray to be specified in a data clause, such as the interior of a matrix. OpenACC requires that data in a data clause be contiguous in memory. For instance, in the PGI Accelerator model, the following is legal:

```
subroutine sub(a,n,m)
 integer :: n, m
 real :: a(n,m)
 integer :: i,j
 !$acc region do copy( a(2:n-1,2:m-1) )
   do j = 2, m-1
     do i = 2, n-1
       a(i,j) = exp(a(i,j))
     enddo
   enddo
end subroutine
```

In OpenACC, the corresponding example would have to move a contiguous subarray, even though some of the elements moved are not used:

```
subroutine sub(a,n,m)
 integer :: n, m
 real :: a(n,m)
 integer :: i,j
 !$acc kernels loop copy( a(1:n,2:m-1) )
   do j = 2, m-1
     do i = 2, n-1
       a(i,j) = exp(a(i,j))
     enddo
   enddo
end subroutine
```

- The PGI Accelerator model allows for two dimensional dynamic C arrays, such as C `float**` arrays, to be moved to the accelerator. OpenACC does not currently allow this, because of the contiguous data requirement.

# Mapping PGI Accelerator Features to OpenACC

Most of the PGI Accelerator Model features map directly to OpenACC features, with some important differences described in the next section.

- The general pragma and directive syntax are the same, with the same `acc` prefix.
- The PGI Accelerator `region` construct maps directly to the OpenACC `kernels` construct.
- The PGI Accelerator `data region` construct maps directly to the OpenACC `data` construct.
- The PGI Accelerator `copy`, `copyin` and `copyout` data clauses map almost directly to OpenACC, with some differences noted in the next section.
- The PGI Accelerator `local` data clause maps to the OpenACC `create` data clause.

- The PGI Accelerator `deviceptr` data clause for C maps to the OpenACC `deviceptr` data clause for C.
- The PGI Accelerator `update device` and `update host` data clauses map directly to OpenACC.
- The PGI Accelerator `async` clause, `wait` directive and async API routines were defined to use an opaque handle. The OpenACC `async` clause, `wait` directive and async API routines use an integer expression. In this release, we have implemented the OpenACC specification for both OpenACC and the PGI Accelerator model.
- The PGI Accelerator `for` (C) and `do` (Fortran) directives map directly to the OpenACC `loop` directive.

# Other Differences

There are four other major differences

- In the PGI Accelerator model, if no data region or data clause for an array is specified at an accelerator compute region, the compile will use `copyin`, `copyout` or `copy` for that array, depending on whether the array is read-only, written-only, or may be partially written or read before written. OpenACC will use `present_or_copyin`, `present_or_copyout` or `present_or_copy`. The PGI implementation will use the OpenACC default when the `-acc` flag is set.
- OpenACC requires that arrays in a data clause be contiguous in memory. This means that subarrays must include whole columns (in C) or whole rows (in Fortran). The PGI implementation will allocate and move contiguous subarrays when the `-acc` flag is set.
- OpenACC has an explicit `reduction` clause for loops, the kernels constructs and the parallel construct. The PGI Accelerator model depends on the compiler to detect reduction operations in the code. This release of the PGI implementation does not recognize the `reduction` clause, but will continue to automatically detect and generate code for reductions.
- OpenACC has an explicit cache directive inside of loop. The PGI Accelerator model has a cache clause on the loop (`for` or `do`) construct, which is treated as a hint to the compiler. This release of the PGI implementation does not recognize the `cache` directive.

# Using the PGI Accelerator Model with OpenACC

There are two ways to use the PGI Accelerator model with OpenACC. One way is to build some of your object files using the PGI Accelerator model directives, compiled with `-ta=nvidia`, and build other objects using OpenACC, compiled with `-acc`. The two sets of object files may linked into a single executable by specifying either `-acc` or `-ta=nvidia` on the link line. The two sets of object files will work in the same program, with the PGI Accelerator routines using the PGI Accelerator model interpretation, and the OpenACC routines using the OpenACC interpretation. However, arrays moved to the GPU in data regions implemented in the PGI Accelerator routines will not be available with the `present` clause in OpenACC routines.

The other way is to recompile PGI Accelerator routines using with the `-acc` flag. This tells the compiler to use the OpenACC interpretation for any features common with OpenACC:

- C subarrays in data clauses, such as `a[2:N]`, will be interpreted using as with OpenACC, meaning `a[2]` and the next `N-1` elements, instead of the PGI Accelerator meaning of `a[2]` through `a[N]`.
- Only contiguous data will be allocated and moved to the accelerator. If you specify a noncontiguous subarray, the runtime will allocate the bounding box containing that subarray.

Even with the `-acc` flag, PGI Accelerator features that extend, but do not conflict with OpenACC features are available, such as `mirror`, `reflected`, and `update` data clauses.

CHAPTER 5

# Implemented Features

This chapter lists the OpenACC features available in this release, and what features will be implemented in upcoming PGI releases.

## In This Release

The following OpenACC features are available in this release:

- The `kernels` construct and the `kernels loop` combined construct.
- The `parallel` construct and `parallel loop` combined construct, in limited cases.
- The `data` construct.
- The `if` clause on the `kernels` and `data` constructs.
- The `async` clause on the `kernels` and `data` constructs.
- The `copy`, `copyin`, `copyout`, `create`, `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout` and `present_or_create` data clauses in C and Fortran.
- The `pcopy`, `pcopyin`, `pcopyout` and `pcreate` alternate spellings for `present`, `present_or_copy`, `present_or_copyin`, `present_or_copyout` and `present_or_create` data clauses.
- The `deviceptr` data clause for C.
- The `loop` construct, and the `seq`, `gang`, `vector`, `independent` and `private` clauses for the loop construct.
- The `update` directive, and all its clauses.
- The `wait` directive.
- Implicit data regions.

- The `declare` directive, except for the `acc_resident` clause.
- The `openacc.h` header file for C.
- The `openacc_lib.h` header file and `openacc` module for Fortran.
- All the runtime API library routines, except `acc_shutdown`.
- The environment variables `ACC_DEVICE_TYPE` and `ACC_DEVICE_NUM`.
- The `_OPENACC` preprocessor macro.

# In Future Releases

The following Open ACC features are not implemented in this release; they will appear in future releases:

- The `parallel` construct and `parallel loop` combined construct, and the clauses specific to the `parallel` construct.
- The `reduction` clause on the kernels construct. The compiler will automatically recognize many reductions.
- The `host_data` construct.
- The `deviceptr` data clause for Fortran dummy arguments.
- The `collapse`, `reduction` and `worker` clauses on the loop construct.
- The `cache` construct.
- The `acc_shutdown` runtime routine.
- The `acc_resident` clause on the `declare` directive.